23 | Julia编译器（二）：如何利用LLVM的优化和后端功能？

2020-07-24 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 14:55 大小 13.67M

你好，我是宫文学。

上一讲，我给你概要地介绍了一下 Julia 这门语言，带你一起解析了它的编译器的编译过程。另外我也讲到，Julia 创造性地使用了 LLVM，再加上它高效的分派机制，这就让一门脚本语言的运行速度，可以跟 C、Java 这种语言媲美。更重要的是，你用 Julia 本身，就可以编写需要高性能的数学函数包，而不用像 Python 那样，需要用另外的语言来编写（如 C 语言）高性能的代码。

那么今天这一讲，我就带你来了解一下 Julia 运用 LLVM 的一些细节。包括以下几个核心要点：

如何生成 LLVM IR？

如何基于 LLVM IR 做优化？

如何利用内建（Intrinsics）函数实现性能优化和语义个性化？

这样，在深入解读了这些问题和知识点以后，你对如何正确地利用 LLVM，就能建立一个直观的认识了，从而为自己使用 LLVM 打下很好的基础。

好，首先，我们来了解一下 Julia 做即时编译的过程。

即时编译的过程

我们用 LLDB 来跟踪一下生成 IR 的过程。

复制代码

```
$ lldb                          #启动lldb
(lldb)attach --name julia       #附加到julia进程
c                               #让julia进程继续运行
```

首先，在 Julia 的 REPL 中，输入一个简单的 add 函数的定义：

复制代码

```
julia> function add(a, b)
            x = a+b
            x
        end
```

接着，在 LLDB 或 GDB 中设置一个断点“br emit_funciton”，这个断点是在 codegen.cpp 中。

复制代码

```
(lldb) br emit_function     #添加断点
```

然后在 Julia 里执行函数 add：

复制代码

```
julia> add(2,3)
```

这会触发 Julia 的编译过程，并且程序会停在断点上。我整理了一下调用栈的信息，你可以看看，即时编译是如何被触发的。

源代码	函数	解释
...		
REPL.jl	japi1_eval_user_input_13306	在REPL中读取用户的输入，并编译和执行
gf.c	jl_apply_generic	gf.c是处理对通用函数的调用 一个函数有多个方法，gf.c要负责做方法的快速分派（dispatch）
...		
toplevel.c	jl_toplevel_eval*	对顶层函数求值 在REPL中求值的都是顶层函数
interpreter.c	jl_interpret_toplevel_thunk eval_body eval_stmt_value eval_value do_call	interpreter.c是解释器 解释器做几件事情：如果你只是在REPL中打印变量或常量的值、给变量赋个常量值等等简单的操作，直接解释执行就行了，不需要即时编译；而当调用一个函数（do_call）的时候，就会调用即时编译的版本
...		
gf.c	jl_apply_generic _jl_invoke jl_compile_method_internal	到gf.c中去分派一个方法执行，发现对应当前参数签名的方法还没被编译，于是启动编译过程
codegen.cpp	jl_compile_linfo emit_function	把函数从Julia的IR编译成LLVM的IR

通过跟踪执行和阅读源代码，你会发现 Julia 中最重要的几个源代码：

gf.c：Julia 以方法分派快速而著称。对于类似加法的这种运算，它会有上百个方法的实现，所以在运行时，就必须能迅速定位到准确的方法。分派就是在 gf.c 里。

interpreter.c：它是 Julia 的解释器。虽然 Julia 中的函数都是即时编译的，但在 REPL 中的简单的交互，靠解释执行就可以了。

codegen.cpp：生成 LLVM IR 的主要逻辑都在这里。

我希望你能自己动手跟踪执行一下，这样你就会彻底明白 Julia 的运行机制。

Julia 的 IR：采用 SSA 形式

在上一讲中，你已经通过 @code_lowered 和 @code_typed 宏，查看过了 Julia 的 IR。

Julia 的 IR也经历了一个发展演化过程，它的 IR 最早不是 SSA 的，而是后来才改成了 SSA 形式。这一方面是因为，SSA 真的是有优势，它能简化优化算法的编写；另一方面也能看出，SSA 确实是趋势呀，我们目前接触到的 Graal、V8 和 LLVM 的 IR，都是 SSA 格式的。

Julia 的 IR 主要承担了两方面的任务。

第一是类型推断，推断出来的类型被保存到 IR 中，以便于生成正确版本的代码。

第二是基于这个 IR 做一些优化，其实主要就是实现了内联优化。内联优化是可以发生在比较早的阶段，你在 Go 的编译器中就会看到类似的现象。

你可以在 Julia 中写两个短的函数，让其中一个来调用另一个，看看它所生成的 LLVM 代码和汇编代码是否会被自动内联。

另外，你还可以查看一下传给 emit_function 函数的 Julia IR 是什么样子的。在 LLDB 里，你可以用下面的命令来显示 src 参数的值（其中，`jl_(obj)`是 Julia 为了调试方便提供的一个函数，它能够更好地显示 Julia 对象的信息，注意显示是在 julia 窗口中）。src 参数里面包含了要编译的 Julia 代码的信息。

复制代码

```
1 (lldb) expr jl_(src)
```

为了让你能更容易看懂，我稍微整理了一下输出的信息的格式：

```
Core.CodeInfo(
    code=Array{Any, (2,)}[
        Expr(:call, Base.add_int, Core.SlotNumber(id=2), Core.SlotNumber(id=3)),
        Expr(:return, SSAValue(1))],
    codelocs=Array{Int32, (2,)}[3, 2],
    ssavaluetypes=Array{Any, (2,)}[Int64,Any],
    ssaflags=Array{UInt8, (2,)}[0x00, 0x00],
    method_for_inference_limit_heuristics=nothing,
    linetable=Array{Core.LineInfoNode, (3,)}[
        Core.LineInfoNode(method=:add, file=Symbol("REPL[1]"), line=2, inlined_at=0),
        Core.LineInfoNode(method=:add, file=Symbol("REPL[1]"), line=3, inlined_at=0),
        Core.LineInfoNode(method=:+, file=Symbol("int.jl"), line=53, inlined_at=1)],
    slotnames=Array{Symbol, (3,)}[Symbol("#self#"),:a,:b],
    slotflags=Array{UInt8, (3,)}[0x00, 0x08, 0x08],
    slottypes=Array{Any, (3,)}[Core.Compiler.Const(val=typeof(Main.add)(), actual=false), Int64,Int64],
    rettype=Int64,
    parent=add(Int64, Int64),
    edges=nothing,
    min_world=0x00000000000012f5,
    max_world=0xffffffffffffffff,
    inferred=true,
    inlineable=true,
    propagate_inbounds=false, pure=false
)
```

两个语句：一个是add，一个是return

三个变量：其中第一个是对函数自身的引用

三个变量的类型

你会发现，这跟用 **@code_typed(add(2,3)) 命令**打印出来的信息是一致的，只不过宏里显示的信息会更加简洁：

```
julia> @code_typed(add(2,3))
CodeInfo(
1 — %1 = Base.add_int(a, b)::Int64
└──      return %1
) => Int64
```

接下来，查看 emit_function 函数，你就能够看到生成 LLVM IR 的整个过程。

生成 LLVM IR

LLVM 的 IR 有几个特点：

第一，它是 SSA 格式的。

第二，LLVM IR 有一个类型系统。类型系统能帮助生成正确的机器码，因为不同的字长对应的机器码指令是不同的。

第三，LLVM 的 IR 不像其他 IR，一般只有内存格式，它还有文本格式和二进制格式。你完全可以用文本格式写一个程序，然后让 LLVM 读取，进行编译和执行。所以，LLVM 的 IR 也可以叫做 LLVM 汇编。

第四，LLVM 的指令有丰富的元数据，这些元数据能够被用于分析和优化工作中。

基本上，生成 IR 的程序没那么复杂，就是用简单的语法制导的翻译即可，从 AST 或别的 IR 生成 LLVM 的 IR，属于那种比较幼稚的翻译方法。

采用这种方法，哪怕一开始生成的 IR 比较冗余，也没有关系，因为我们可以在后面的优化过程中继续做优化。

在生成的 IR 里，会用到 Julia 的**内建函数**（Intrinsics），它代表的是一些基础的功能。

在JavaScript 的编译器里，我们已经接触过**内置函数**（Built-in）的概念了。而在 Julia 的编译器中，内建函数和内置函数其实是不同的概念。

内置函数是标准的 Julia 函数，它可以有多个方法，根据不同的类型来分派。比如，取最大值、最小值的函数 max()、min() 这些，都是内置函数。

而内建函数只能针对特定的参数类型，没有多分派的能力。Julia 会把基础的操作，都变成对内建函数的调用。在上面示例的 IR 中，就有一个 add_in() 函数，也就是对整型做加法运算，它就是内建函数。内建函数的目的是生成 LLVM IR。Julia 中有近百个内置函数。在intrinsics.cpp中，有为这些内置函数生成 LLVM IR 的代码。

这就是 Julia 生成 LLVM IR 的过程：遍历 Julia 的 IR，并调用 LLVM 的 IRBuilder 类，生成合适的 IR。在此过程中，会遇到很多内建函数，并调用内建函数输出 LLVM IR 的逻辑。

运行 LLVM 的 Pass

我们之所以会使用 LLVM，很重要的一个原因就是利用它里面的丰富的优化算法。

LLVM 的优化过程被标准化成了一个个的 Pass，并由一个 PassManager 来管理。你可以查看 jitlayers.cpp 中的addOptimizationPasses() 函数，看看 Julia 都使用了哪些 Pass。

Pass名称	描述
CFGSimplification	合并基本块、删除不可达基本块、简化终结指令（如return等指令）、把switch转化为查找表等等
SROA	把聚集量（如struct、vector等）转化为标量
InstructionCombining	把多条指令合并成一条指令。比如，“%1=%x+1 %2=%1+1”两条指令，改成一条指令“%1=%x+2”
EarlyCSE	在Dominator tree上做一次简单的快速的公共子表达式消除
MemCpyOpt	消除不必要的内存拷贝(MemCpy)调用，以及把多个写内存的操作简化为MemSet操作 MemCpy和MemSet都是LLVM的Intrinsics，它调用C标准库，对内存做批量操作，性能更高 举例子来说，你可能用一个循环语句，把一个数组中的n个元素全部初始化为某个值，而这个操作如果用MemSet，性能会更高 所以，你经常可以见到LLVM把代码优化成对MemCpy、MemSet的调用
AlwaysInliner	处理那些标记为“总是内联”的函数
ScopedNoAliasAnalysis	带作用域的无别名分析，这是别名分析的一种 基于LLVM IR的元数据信息
TypeBasedAliasAnalysis	又称TBAA，基于类型信息做别名分析 它的原理简单地可以描述为：如果两个指针的类型都不一样，那么它们应该不是指向相同地址的
BasicAA	基础别名分析，其分析结果永远不会被无效（never-invalidated）
DeadCodeElimination	死代码删除
Reassociate	代码重组 这个优化是把表达式改写为等价的格式，但更有利于做其他优化，如常数传播、全局公共子表达式消除（GCSE）、循环不变代码移动（LICM）、部分冗余消除（PRE）等等 比如：4 + (x + 5) 优化为 x + (4 + 5)
LoopRotate	旋转循环，参考官方文档对它的解释
LICM	循环不变代码移动
LoopUnswitch	消除循环中的分支（比如把if...else提到loop外面）
InductionVariableSimplify	对循环中的归纳变量的简化
LoopDeletion	消除可认定为死代码的循环
SimpleLoopUnroll	一个简单的循环展开优化
GVN	全局值编号优化
SCCP	稀疏有条件的常量传播
DeadStoreElimination	死存储消除 死存储指的是一个变量存了一个数据到内存，但永远没有被用到
SLPVectorizer	SLP是Superword-Level Parallelism的缩写 LLVM中有多个向量化优化器，SLP是对基本块的代码做向量化优化 做矢量化的对象是不同的指令，但可以并在一起执行，减少指令次数，可以参考官方文档中的解读
LoopVectorizer	循环矢量化，这是另一个矢量优化器 它的原理是，对同一个指令，循环多次执行。向量化之后，减少循环次数，而每个循环中由于使用了矢量化指令（SIMD指令），效果相当于同一个指令执行了多次 可以参照《编译原理之美》第31讲中的例子
AggressiveDCE	激进的DCE（死代码消除） 这个Pass使用了基于SSA的激进DCE算法，该算法假设所有指令都是死的，除非有相反证据，这使得它能够消除不那么明显的死代码

LoopRotate：官方文档 / SLPVectorizer：官方文档 /《编译原理之美》：第 31 讲

上面表格中的 Pass 都是 LLVM 中自带的 Pass。你要注意，运用好这些 Pass，会产生非常好的优化效果。比如，某个开源项目，由于对性能的要求比较高，所以即使在 Windows 平台上，仍然强烈建议使用 Clang 来编译，而 Clang 就是基于 LLVM 的。

除此之外，Julia 还针对自己语言的特点，写了几个个性化的 Pass。比如：

Pass名称	描述
CombineMulAdd	把一条Mul指令和一条Add指令，合并成一个FMA指令 FMA是Fused Multiply-Add的缩写，意思是融合了乘法和加法的指令，它能在一条指令中完成a=a*c+b，a=b*a+c，a=b*c+a甚至a=b*c+d的运算，从而提升了计算效率 像AMD64架构的CPU，很多都支持FMA指令集
LowerSIMDLoop	在循环上添加一些元数据信息，使之成为LLVM的平行循环，从而被矢量化
FinalLowerGC GCInvariantVerifier LateLowerGCFrame	设置并验证与垃圾收集有关的指令

这些个性化的 Pass 是针对 Julia 本身的语言特点而编写的。比如对于垃圾收集，每种语言的实现策略都不太一样，因此就必须自己实现相应的 Pass，去插入与垃圾收集有关的代码。再比如，Julia 是面向科学计算的，比较在意数值计算的性能，所以自己写了两个 Pass 来更好地利用 CPU 的一些特殊指令集。

emit_function 函数最后返回的是一个**模块（Module）对象**，这个模块里只有一个函数。这个模块会被加入到一个JuliaOJIT对象中进行集中管理。Julia 可以从 JuliaOJIT 中，查找一个函数并执行，这就是 Julia 能够即时编译并运行的原因。

不过，我们刚才说的都是生成 LLVM IR 和基于 IR 做优化。**那么，LLVM 的 IR 又是如何生成机器码的呢？对于垃圾收集功能，LLVM 是否能给予帮助呢？在使用 LLVM 方面还需要注意哪些方面的问题呢？**

利用 LLVM 的正确姿势

在这里，我给你总结一下 LLVM 的功能，并带你探讨一下如何恰当地利用 LLVM 的功能。

通过这门课，你其实已经能够建立这种认识：编译器后端的工作量更大，某种意义上也更重要。如果我们去手工实现每个优化算法，为每种架构、每种 ABI 来生成代码，那不仅工作量会很大，而且还很容易遇到各种各样需要处理的 Bug。

使用 LLVM，就大大降低了优化算法和生成目标代码的工作量。LLVM 的一个成功的前端是 Clang，支持对 C、C++ 和 Objective-C 的编译，并且编译速度和优化效果都很优秀。既然它能做好这几种语言的优化和代码生成，那么用来支持你的语言，你也应该放心。

总体来说，LLVM 能给语言的设计者提供这样几种帮助：

程序的优化功能

你可以通过 LLVM 的 API，从你的编译器的前端生成 LLVM IR，然后再调用各种分析和优化的 Pass 进行处理，就能达到优化目标。

LLVM 还提供了一个框架，让你能够编写自己的 Pass，满足自己的一些个性化需求，就像 Julia 所做的那样。

LLVM IR 还有元数据功能，来辅助一些优化算法的实现。比如，在做基于类型的别名分析（TPAA）的时候，需要用到在前端解析中获得类型信息的功能。你在生成 LLVM IR 的时候，就可以把这些类型信息附加上，这样有助于优化算法的运行。

目标代码生成功能

LLVM 支持对 x86、ARM、PowerPC 等各种 CPU 架构生成代码的功能。同时，你应该还记得，在第 8 讲中，我说过 ABI 也会影响代码的生成。而 LLVM，也支持 Windows、Linux 和 macOS 的不同的 ABI。

另外，你已经知道，在目标代码生成的过程中，一般会需要三大优化算法：指令选择、寄存器分配和指令排序算法。LLVM 对此同样也给予了很好的支持，你直接使用这些算法就行了。

最后，LLVM 的代码生成功能对 CPU 厂家也很友好，因为这些算法都是**目标独立**（Target-independent）的。如果硬件厂家推出了一个新的 CPU，那它可以用 LLVM 提供的 **TableGen 工具**，来描述这款新 CPU 的架构，这样我们就能使用 LLVM 来为它生成目标代码了。

对垃圾收集的支持

LLVM 还支持垃圾收集的特性，比如会提供安全点、读屏障、写屏障功能等。这些知识点我会在第 32 讲“垃圾收集”的时候带你做详细的了解。

对 Debug 的支持

我们知道，代码的跟踪调试对于程序开发是很重要的。如果一门语言是生成机器码的，那么要实现跟踪调试，我们必须往代码里插入一些调试信息，比如目标代码对应的源代码的位置、符号表等。这些调试信息是符合 **DWARF**（Debugging With Attributed Record Formats，使用有属性的记录格式进行调试）标准的，这样 GDB、LLDB 等各种调试工具，就可以使用这些调试信息进行调试了。

对 JIT 的支持

LLVM 内置了对 JIT 的支持。你可以在运行时编译一个模块，生成的目标代码放在内存里，然后运行该模块。实际上，Julia 的编译器能够像普通的解释型语言那样运行，就是运用了 LLVM 的 JIT 机制。

其他功能

LLVM 还在不断提供新的支持，比如支持在程序链接的时候进行过程间的优化，等等。

总而言之，研究 Julia 的编译器，就为我们使用 LLVM 提供了一个很好的样本。你在有需要的时候，也可以作为参考。

课程小结

今天这一讲，我们主要研究了 Julia 如何实现中后端功能的，特别是在这个过程中，它是如何使用 LLVM 的，你要记住以下要点：

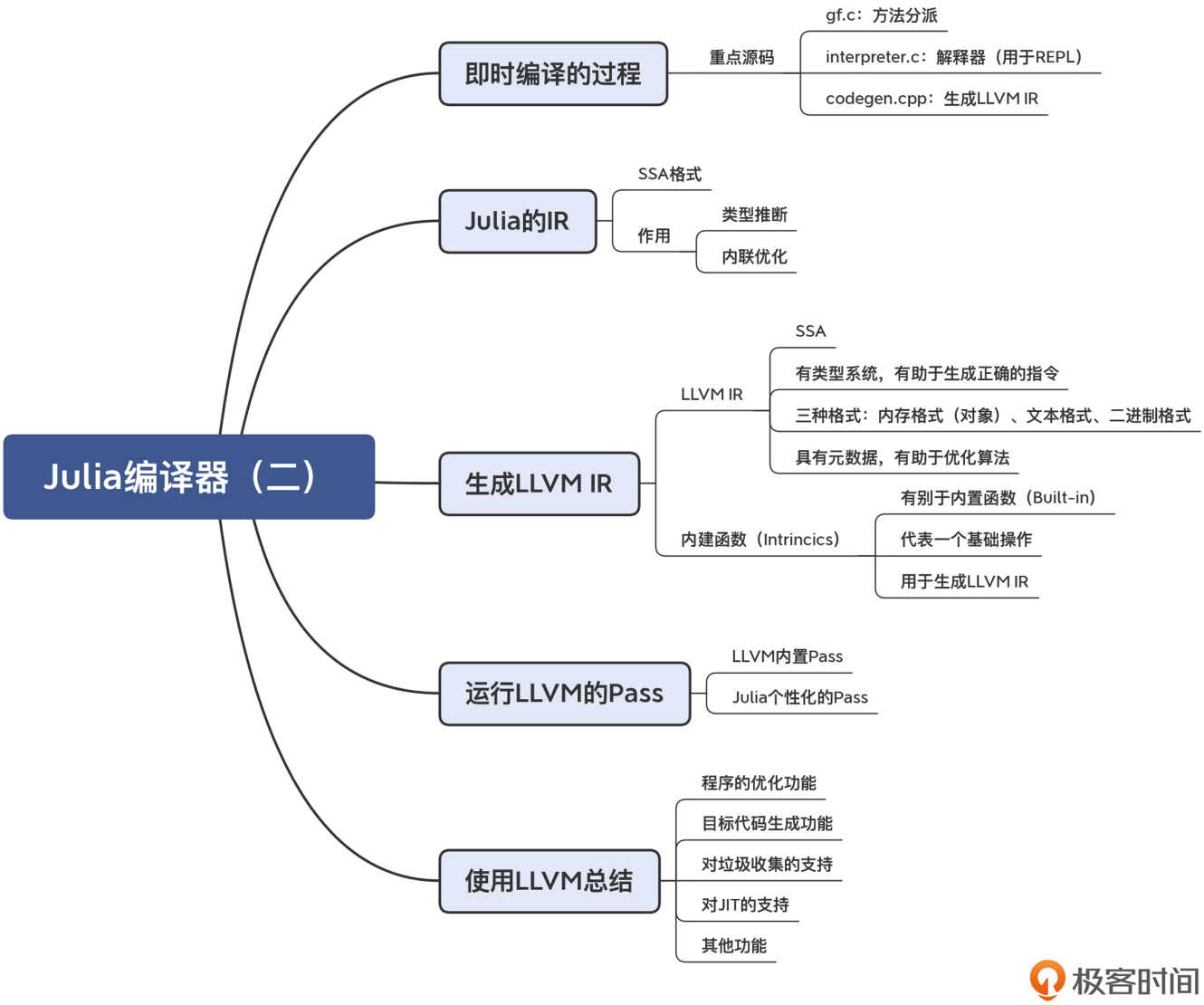
Julia 自己的 IR 也是采用 **SSA 格式**的。这个 IR 的主要用途是**类型推断和内联优化**。

Julia 的 IR 会被**转化成 LLVM 的 IR**，从而进一步利用 LLVM 的功能。在转换过程中，会用到 Julia 的**内建函数**，这些内建函数代表了 Julia 语言中，抽象度比较高的运算功能，你可以拿它们跟 V8 的 IR 中，代表 JavaScript 运算的高级节点作类比，比如加法计算节点。这些内建函数会生成体现 Julia 语言语义的 LLVM IR。

你可以使用 **LLVM 的 Pass 来实现代码优化**。不过使用哪些 Pass、调用的顺序如何，是由你自己安排的，并且你还可以编写自己个性化的 Pass。

LLVM 为程序优化和生成目标代码提供了可靠的支持，值得重视。而 Julia 为使用 LLVM，就提供了一个很好的参考。

本讲的思维导图我也给你整理出来了，供你参考和复习回顾知识点：



一课一思

LLVM 强调全生命周期优化的概念。那么我们来思考一个有趣的问题：能否让 Julia 也像 Java 的 JIT 功能一样，在运行时基于推理来做一些激进的优化？如何来实现呢？欢迎在留言区发表你的观点。

参考资料

1. LLVM 的官网：llvm.org。如果你想像 Julia、Rust、Swift 等语言一样充分利用 LLVM，那么应该会经常到这里来查阅相关资料。
2. LLVM 的源代码。像 LLVM 这样的开源项目，不可能通过文档或者书籍来获得所有的信息。最后，你还是必须去阅读源代码，甚至要根据 Clang 等其他前端使用 LLVM 的输出做反向工程，才能掌握各种细节。LLVM 的核心作者也推荐开发者源代码当作文档。
3. Working with LLVM：Julia 的开发者文档中，有对如何使用 LLVM 的介绍。

4. LLVM’ s Analysis and Transform Passes：对 LLVM 中的各种 Pass 的介绍。要想使用好 LLVM，你就要熟悉这些 Pass 和它们的使用场景。
5. 在《编译原理之美》的第 25 讲和第 26 讲，我对 LLVM 后端及其命令行工具做了介绍，并且还手工调用 LLVM 的 API，示范了针对不同的语法结构（比如 if 结构）应该如何生成 LLVM IR，最后即时编译并运行。你可以去参考看看。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | Julia编译器（一）：如何让动态语言性能很高？

下一篇 24 | Go语言编译器：把它当作教科书吧

精选留言 (1)

写留言

鱼_XueTr
2020-07-26

jinpeng.d@Mac$□ lldb
(lldb) attach --name julia
error: attach failed: could not find a process named julia
(lldb) ^D

展开

1